WWW.JAMK.FI

Käyttöliittymän automaatiotestaus

Jari Hintsala

Opinnäytetyö
Joulukuu 2012

Ohjelmistotekniikan koulutusohjelma
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) HINTSALA, Jari	Julkaisun laji Opinnäytetyö	Päivämäärä 07.12.2012
	Sivumäärä 31	Julkaisun kieli Suomi
	Luottamuksellisuus ( ) saakka	Verkkojulkaisulupa myönnetty (X)
Työn nimi KÄYTTÖLIITTYMÄN AUTOMAATIOTESTAUS		
Koulutusohjelma Ohjelmistotekniikka		
Työn ohjaaja(t) SALMIKANGAS, Esa		
Toimeksiantaja(t) Tietokarhu Oy		

Tiivistelmä

Opinnäytetyössä tutkittiin käyttöliittymän automaatiotestauksen hyötyjä ja onko Microsoft Visual Studio 2010 Coded UI Automated Test –työkalusta hyötyä apuvälineenä suuremmille organisaatioille.

Tutkimuksessa otetaan kantaa automaatiotestien hyötyihin, heikkouksiin ja ylläpitoon. Tutkimuksessa tarkastellaan myös kustannustehokkuutta, resurssointia sekä testien ylläpidettävyyttä. Tutkimuksessa käydään myös läpi testauksen periaatteita, erilaisia testausmenetelmiä sekä luodaan katsaus testauksen historiaan. Tuloksia voidaan pitää yleispätevinä sillä monet käyttöliittymän automaatiotestaustyökalut käyttävät samaa periaatetta kuin testattavana ollut työkalu. Vertailukohteena käytettiin Visual Studio 2012 – version vastaavaa työkalua.

Tutkimuksen tulokset ovat kannustavia ja suosivat käyttämään automaatiotestausta osana testausprosessia. Käyttöliittymän automaatiotestit takaavat tasaisen laadun ja inhimillisten virheiden vaikutus vähenee huomattavasti koska testit toistuvat joka kerta samanlaisina. Automaatiotestaus ei kuitenkaan tule korvaamaan manuaalista testaamista vaan antaa tukea testaukseen ja helpottaa regressiotestausta sekä vähentää manuaalisesti suoritettavaa testausta.

Ohjelmistolla on potentiaalia nousta tärkeäksi osaksi päivittäistä testausta. Ohjelmien vaatimusten muuttuessa jatkuvasti voi testien ylläpidettävyystä tulla ongelmana joka on kuitenkin ratkaistavissa huolellisella suunnittelulla ja riittävällä henkilöstön koulutuksella. Testaajan ja kehittäjän välisellä komminukoinnilla ja tiedon jakamisella on äärimmäisen tärkeä rooli jotta käyttöliittymältä saadaan testattua riittävän tehokkaasti ja monipuolisest. Tutkimuksesta on hyötyä yrityksille jotka suunnittelevat automaatiotestauksen käyttöönottoa.

Avainsanat (asiasanat)

Käyttöliittymän automaatiotestaus, automaatiotestaus, coded UI

Muut tiedot

Author(s)
HINTSALA, Jari

Type of publication
Bachelor´s Thesis

Date
07122012

Pages
31

Language
Finnish

Confidential

( ) Until

Permission for web publication
(X)

Title
AUTOMATED TESTING FOR USER INTERFACE

Degree Programme
Software Engineering

Tutor(s)
SALMIKANGAS, Esa

Assigned by
Tietokarhu Oy

Abstract

The purpose of this bachelor's thesis was to examine the benefits of automated testing of user interface and study the benefits of Microsoft Visual Studio 2010 Coded UI Automated Testing tool for larger organizations.

The research takes a stand on benefits, weaknesses and maintenance of automated tests. The research also goes through cost effectiveness, resources and the maintenance of tests as well as discusses testing principles, different testing methods and history of testing. The results are universal since many automated testing tools for user interface use the same principles than the tested tool. The comparison tool was Visual Studio 2012 version of the same tool.

The results of the research were supportive and suggest using automated testing tools as a part of the testing process. User Interface testing guarantees even quality and human errors do not affect the process very much since the tests are always the same. Automated testing does not replace manual testing but it gives support to testing and eases regression testing and decreases manually executable testing.

The tool has potential to be an important part of daily testing. The software requirements are changing constantly, thus test maintenance is a problem that can be solved with careful planning and sufficient and relevant staff education. The communication between tester and developer and the sharing of knowledge has a very important role since the user interface is to be tested in an effective and versatile enough way. The study is useful for corporations planning to introduce automated testing.

Keywords

User interface automated testing, automated testing, coded UI

Miscellaneous

SISÄLTÖ

KUVIOT

KÄSITTEET

Assertointi

Assertointi on testin toteuttava tarkistusmetodi.

Black Box

Testataan ohjelmaa ilman tietoa tarkemmasta toteutuksesta.

C#

Oliopohjainen ohjelmointikieli.

Debuggaus

Ohjelmaa ajetaan rivi kerrallaan, yleensä etsitään virheitä ja tarkastellaan kuinka ohjelma toimii rivi kerrallaan.

Defekti

Defekti on ohjelmistossa oleva puute tai virhe joka ei vastaa ohjelmalle asetettuja vaatimuksia tai loppukäyttäjän odotuksia.

Metodi

Osa luokan tai olio koodia joka suorittaa jonkin toiminnallisuuden.

Rollback

Rollback on operaatio joka palauttaa tietokannan aikaisempaan tilaansa.

Stub

Stubin avulla voi määritellä jonkin esim. rajapintaa kutsuvan metodin testeissä käytettäväksi ”vale” – metodiksi.

TDD

Test driven development eli kirjoitetaan testit ennen varsinaista ohjelmakoodia.

UI

User Interface eli käyttöliittymä.

Validointi

Tuloksien vertailua haluttuun lopputulokseen.

White Box

Testataan ohjelmaa tietäen miten se on toteutettu.

XML

XML on merkintäkieli tai standardi, jolla tiedon merkitys on kuvattavissa tiedon sekaan. XML-kieltä käytetään sekä formaattina tiedonvälitykseen järjestelmien välillä että formaattina dokumenttien tallentamiseen.

1 TYÖN LÄHTÖKOHDAT

1.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana oli Tietokarhu Oy, joka on Tiedon ja Suomen valtion yhteisyhtiö, jossa työskentelee noin 300 tietotekniikan ammattilaista Helsingin Kalasatamassa sekä Jyväskylän Mattilanniemessä. Tietokarhu kehittää ja ylläpitää Verohallinnon tietojärjestelmiä.

1.2 Opinnäytetyön tavoitteet

Opinnäytetyön tavoitteena oli selvittää miten Visual Studion 2010 Feature pack 2 mukana tullut Coded UI Automated Test – työkalu soveltuu käytännön päivittäiseen työhön. Tarkoituksena oli testata työkalun ominaisuuksia ilman mitään todellista sovellusta. Tietokarhun salaisen työn luonteen vuoksi heidän käyttämiään käyttöliittymiä ei voida käyttää testaukseen vaan tutkimusta varten on rakennettu erillinen testiympäristö. Lisäksi vertaan Visual Studio 2010 ja 2012 versioiden eroja.

Tarkoituksena on testata työkalun soveltuvuutta todellisen työelämän tarkoituksiin, onko se riittävän joustava ja ylläpidettävä suurelle organisaatiolle ja saadaanko sitä käyttämällä riittävä hyötysuhde. Tutustutaan työkalun eri ominaisuuksiin ja tehdä läpikatsaus kuinka työkalulla luodaan testejä, minkälainen rakenne testeissä on ja mitä asioita olisi hyvä ottaa huomioon testejä tehdessä. Työssä tarkastellaan myös kenen kannattaa käyttää työvälinettä ja saadaanko sillä todellisia hyötyjä.

Teoria puolella keskitytään erilaisiin testausmenetelmiin sekä niiden erilaisiin käyttötarkoituksiin ja tutkitaan testausta yleisemmälläkin tasolla. Tutkimuksessa käydään läpi myös erilaisia testausprosesseja ja testaustasoja sekä mitä kullakin tasolla on tarkoitus tehdä ja kuinka viedään läpi tyypillinen testausprosessi. Tutkimuksessa tarkstellaan myös mitkä testauksen osa-alueet ovat testaajan ja mitkä kehittäjän vastuulla sekä otetaan lyhyt katsaus testauksen historiaan.

2 TESTAUS YLEISESTI JA TESTAUSMALLIT

2.1 Testauksen periaatteet

Ohjelmistojen testaus on mitä tahansa toimintaa joka tähtää kehittämään ohjelmiston kykyä kohdata ohjelmistolle asetetut vaatimukset. Testauksella on vaikeaa havaita kaikkia virheitä järjestelmässä. Kaikkien virheiden löytyminen vaatisi paljon resursseja ja siitä saatava hyöty ei olisi riittävän suuri.

Ohjelmistotestauksen pääperiaatteena on löytää virheet järjestelmässä. Testit eivät voi varmistaa ohjelmiston toimintaa kaikissa olosuhteissa mutta voivat varmistaa että ohjelma toimii määritetyissä olosuhteissa oikein. (Pan chpt.1)

2.2 Testauksen historia

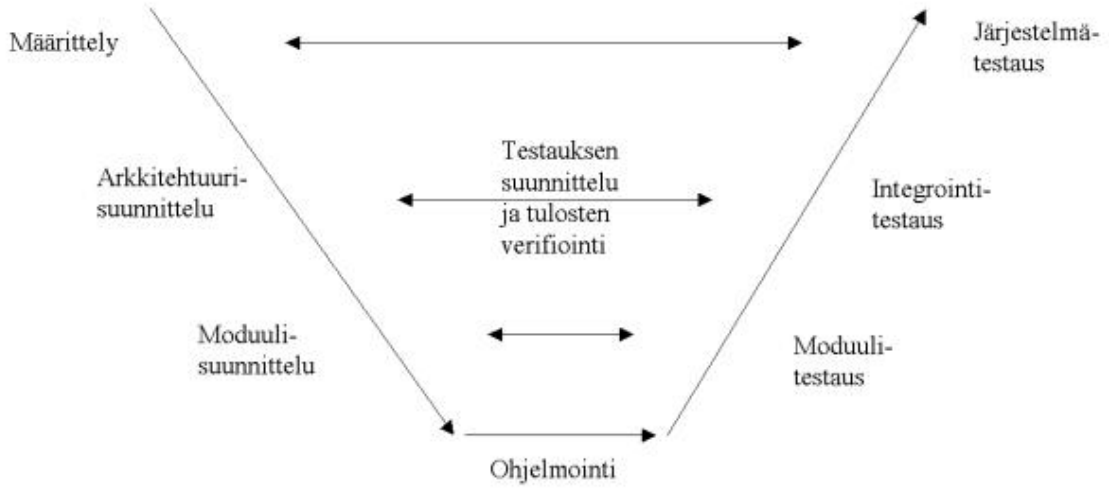
Ohjelmia on testattu niin kauan kuin ohjelmia on ollut olemassa, testaamisen konsepti on vain muuttunut ajan myötä. Ohjelmistotestauksen historia voidaan karkeasti jakaa viiteen eri vaiheeseen:

- Ensimmäinen vaihe oli aika ennen vuotta 1956, jolloin testaaminen oli sama asia kuin debuggaus.
- Toinen vaihe sijoittuu aikavälille 1957–1978 joka oli aika jolloin testattiin että ohjelma vastaa sille asetettuja spesifikaatioita.
- Kolmas vaihe oli 1979–1982, jolloin pääasiallisesti testattiin toteutuksessa olevia virheitä.
- Neljäs vaihe oli 1983–1987, jolloin pyrittiin löytämään virheitä vaatimuksissa, suunnittelussa ja toteutuksessa.
- Viides ja viimeisin vaihe alkoi 1988. Testauksen tarkoituksena on virheiden ehkäisy vaatimuksissa, suunnittelussa ja toteutuksessa. (Luo)

2.3 V-malli

V-malli määrittelee kuka tekee mitä tekee ja milloin tekee. V-mallia voidaan muokata projektin tarpeiden mukaan, jolloin sen rakenne saattaa hieman muuttua.

V-malli kehitettiin ratkaisemaan vesiputousmallin ongelmat. Virheet löytyivät vesiputousmallissa liian myöhäisessä testauksen vaiheessa. Kuviossa 1 esitetään kuinka V-malli lähtee liikkeelle suunnittelusta jossa ensin tehdään vaatimusmäärittely. Tämän jälkeen siirrytään suunnitteluvaiheeseen, jonka aikana suunnitellaan tuote aloittaen laajemmasta konseptista siirtyen tarkempaan esimerkiksi yksittäisiin toimintoihin. Seuraava vaihe on tuotteen toteutus. Kolmas eli viimeinen vaihe V-mallissa on testaus, joka suoritetaan tarkemmasta yleisempään tasoon. Tässä vaiheessa testataan esimerkiksi ensimmäisenä tuotteen toiminnot, sitten koko tuotteen toiminta ja tämän jälkeen testaus tapahtuu loppukäyttäjän toimesta.

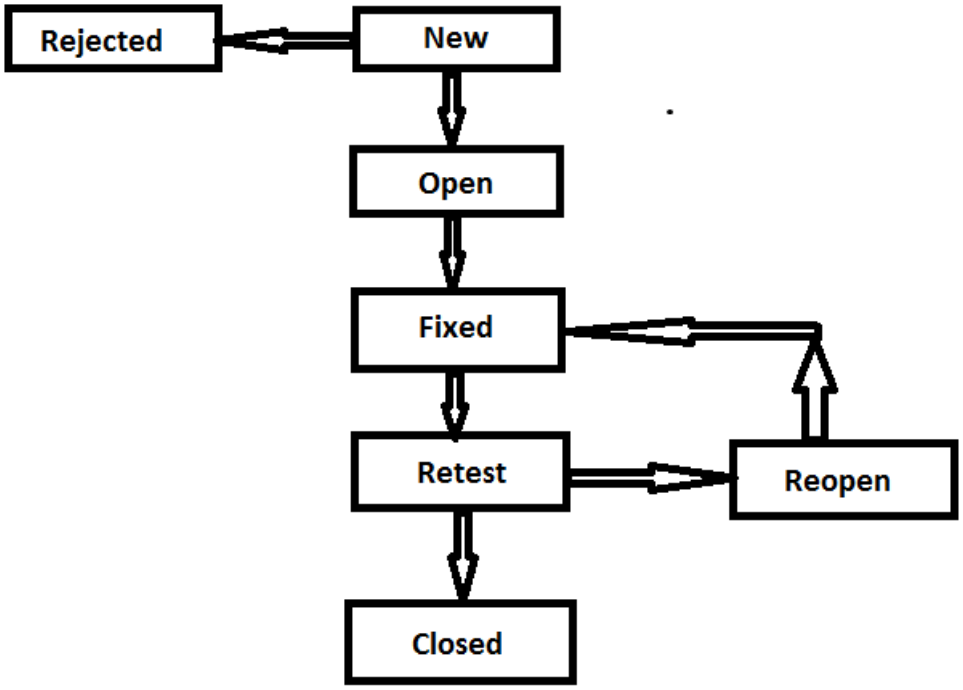


KUVIO 1 V-malli (Testaussuunnitelma)

Testausvaiheissa verrataan jatkuvasti vastaako tuote samalla tasolla olevaa suunnitteluvaihetta. Eli viimeistä testausta verrataan vaatimusmäärittelyyn, joka oli suunnittelun ensimmäinen vaihe. V-mallissa testauksen luonne muuttuu siirryttäessä alimmalta tasolta ylöspäin White Box – testauksesta Black Box – testaukseen. (Graham & Veenendaal s.36–38, SPM)

3 OHJELMISTOVIRHEEN ELINKAARI

Ohjelmiston suorittaessa testauskierroksiaan läpi, on mahdollista että virheitä tai puutteita löytyy testattavasta tuotteesta. Virheeksi voidaan luetella ohjelmassa esiintyvät puutteet, ongelmat tai bugit. Oikeastaan ohjelmistovirhe voi olla mikä vain ongelma tai puute ohjelmassa, nimeäminen vaihtelee riippuen organisaatiossa käytettävistä menetelmistä ja nimeämiskäytännöistä. Ohjelmistovirhe on määriteltävä tarkasti organisaatiossa jotta testaajat tietävät testatessaan mikä todellisuudessa on virhe ja mikä ei. Kuviossa 2 esitetään ohjelmistovirheen elinkaaren eri tilat.



KUVIO 2 Ohjelmistovirheen elinkaari (Defect Management)

Yleisesti ohjelmistovirheen määritelmä on että se on oltava toistettavissa. Kehittäjälle on viisasta antaa tieto virheen aiheuttaneista toimenpiteistä kohta kohdalta, jottei aikaa kehittäjällä mene aikaa hukkaan yrittäessään saada toistumaan samankaltaista tilannetta joka on testaajalla jo selvästi tiedossa. Testaajan on varmistettava onko virhe uusi vai jo aiemmin todettu virhe. Ohjelmistovirheen ollessa jo olemassa on tärkeää

ettei samaa virhettä kirjata enään uudelleen. Jos virhe tapahtuu uudelleen, vaikka sen pitäisi olla jo korjattuna, tulee sen tila muuttaa ja asettaa virhe uudelleen avatuksi.

Kaikki ohjelmistovirheet on varmistettava oikeiksi ja tiimin on hyvä käydä läpi virheitä ja katsoa minkälaisia toimenpiteitä ne vaativat. Ohjelmistovirheen elinkaaren alkuvaiheessa virhe tai puute analysoidaan ja tarkistetaan ettei virhe ole sama kuin jokin on aiemmin korjattu, jolloin virheen status muuttuisi ”New” – tilasta ”Reopen” – tilaan, eikä ”Open” – tilaan kuten normaalisti. Jos virhe on uusi, on testaajan annettava kaikki mahdollinen informaatio virheestä kehittäjälle.

Kehittäjän ottaessa ohjelmistovirheen käsittelyyn sen tilaksi tulee asettaa ”Open”. Kehittäjän tulisi kirjoittaa kuinka aikoo virheen korjata ja mikä tulee olemaan korjauksen lopputulos. Kun virhe on korjattu, yksikkötestattu ja paritarkastettu asetetaan se ”Fixed” – tilaan.

Seuraavaksi testaaja ottaa ohjelmistovirheen testaukseen ja korjauksen ollessa onnistunut muuttaa virheen ”Closed” – tilaan. Mikäli virheen korjauksesta löytyy edelleen puutteita muutetaan sen tilaksi ”Reopen”. Tämän jälkeen virhe lähtee uudelle korjauskierrokselle kunnes testaaja voi muuttaa tilaksi ”Closed”, korjauksen ollessa valmis. (Limaye s.207–208)

4 YKSIKKÖTESTAUS

4.1 Yksikkötestauksen periaatteet

Yksikkötestauksessa on ideana ottaa pienen mahdollinen pala testattavasta ohjelmasta, jonka voi eristää koodista ja päätellä toimiiko se halutulla tavalla. Jokainen yksikkö tai lohko koodista tulisi olla testattuna. Yksikkötestit paljastavat suuren osan ohjelmavirheistä ennen kuin pääsevät siirtymään seuraavalle testaustasolle asti. (MSDN)

Periaatteena yksikkötestauksessa on varmistaa että jokainen ohjelman osa vastaa sille asetettuja määrityksiä. Testitapaukset voidaan luoda käyttäen joko Black Box- tai White Box – testausmenetelmiä. Yleensä yksikkötestausta suoritetaan käyttäen White Box – testausta. (Brunstein s.138)

Yksikkötestaus suoritetaan testaustasolla ennen integraatiotestausta. Tyypillisesti yksikkötestin tekee ohjelmistokehittäjä itse, jonka on mahdollista ymmärtää yksittäisen osien toimintaa testauksessa.

Yksikkötestit lisäävät luottamusta siihen että muutosten/ylläpidon jälkeen ohjelmisto toimii yhä halutulla tavalla. Oikein kirjoitettu yksikkötesti huomaa jos koodi ei toimi enää muutosten jälkeen kuten pitäisi ja kehittäjän on helppo selvittää mistä virhe voisi johtua. Lisäksi kehittäminen on nopeampaa sillä ohjelmaa voi testata yksikkötestin avulla menemättä aina käyttöliittymän kautta katsomaan onko jokin muutos saatu toimimaan.

Tällä tasolla korjaaminen myös tulee huomattavasti halvemmaksi kuin testauksen ylemmillä tasoilla, jolloin kaikki pitää kierrättää uudelleen testaajalta kehittäjälle ja takaisin. (STF)

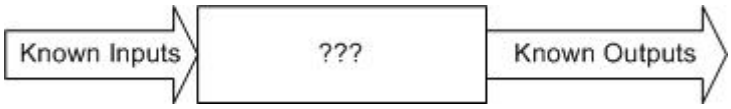
4.2 Koodikattavuus

Koodikattavuutta käytetään määrittelemään kuinka laajasti testit suorittavat ohjelmakoodia. Tieto helpottaa päättelemään kuinka tehokkaita testejä on kirjoitettu. Tämän avulla saadaan selville mitkä sektiot koodista testit kattavat ja näiden tietojen avulla voi päätellä, minkälaisia testejä olisi hyvä kirjoittaa lisää. (MSDN)

Koodikattavuutta tarkastellessa on vaarana se että vaikka mittaustulos osoittaisi että kaikki koodin lohkot on täysin testattu, ei se sitä välttämättä ole. Koodikattavuus mittaa mitkä kaikki lohkot on käyty läpi mutta syötearvoja tulisi olla testattavana useampi kuin vain yksi. Koodikattavuus ei pysty tunnistamaan tätä, vaan ilmoittaa että kaikki koodi on täysin testattua vaikka todellisuudessa näin ei olekaan. Testattuna onkin vain yhdellä syötearvolla testattuna, että tapaus toimii ja on testattu. (Graham & Veenendaal s.105–107)

4.3 Black Box

Black Box -testaus tunnetaan myös funktionaalisena testauksena jossa ohjelmaa testataan syöttämällä arvoja ja katsomalla lopputulosta tutkimatta mitä ohjelman sisällä tapahtuu. Kuviossa 3 esitetään kuvallisesti miten Black Box – testaus etenee.



KUVIO 3 Black Box –testaus (Codeproject)

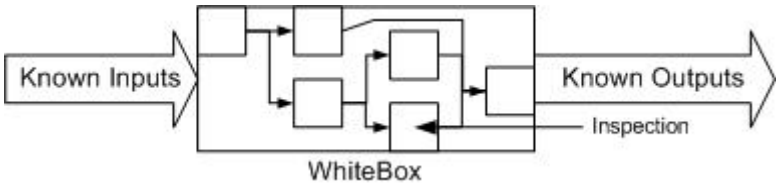
Black Box -testaus rajoittaa kykyä testata koko ohjelmistokoodia läpi, pääasiallisesti sen vuoksi ettei se tutki kaikki koodin mahdollisia polkuja ja ehtolauseita. Tyypillisesti Black Box -testaus varmistaa että syötetyille arvoille saadaan aikaan haluttu lopputulos.

Luokat ovat yleensä Black Box -testauksessa toteutettu siten että luokassa on paljon julkisia metodeita joita päästään testaamaan suoraan. Yleensä ohjelmakoodia testataan raja-arvoilla sekä rajatulla määrällä oikeita sekä virheellisiä syötearvoja.

Black Box – testauksen hyötyjä ovat mm. se että testaaja ei tarvitse ohjelmointitaitoja eikä testaajan tarvitse tietää kuinka ohjelma on rakennettu. Haittapuolena mainittakoon se että suuri osa mahdollisia syötteitä jää testaamatta ja testit voivat olla toisteisia, koska kehittäjä on saattanut testata asian jo kertaalleen yksikkötesteillä. (Kassem s.224–226, STF)

4.4 White Box

White Box – testaus on testausmetodi jossa ohjelmanrakenne sekä toteutus ovat testaajan tiedossa, kuviossa 4 näytetään kuinka White Box – testaus etenee. Testaaja tietää mitä polkuja pitkin haluaa ohjelmakoodin etenevän ja valitsee syötteet sen mukaisesti. Ohjelmointitaidot ovat välttämättömät White Box – testauksessa.



KUVIO 4 White Box –testaus (Codeproject)

White Box -testauksessa testaajalla on oltava mahdollisuus tietoon jolla edetä koko koodi kaikkine polkuineen. Testauksessa käytetään oikeita sekä vääriä syötearvoja, jolloin saadaan testattua että virhetapaukset tulee käsiteltyä oikein.

Muutamia White Box – testauksen hyötyjä ovat mm. se että testit voidaan suorittaa aikaisemmassa vaiheessa. Ei tarvitse odottaa että käyttöliittymä on täysin valmis sekä testit ovat paljon kattavampia koska koodipolkuja osataan kulkea tarkemmin.

Haasteena tämän kaltaisessa testauksessa on testien monimutkaisuus, jolloin testaajan tulee olla todella asiansa tunteva. Testaus voi muuttua raskaaksi jos toteutus muuttuu jatkuvasti.

Koodin täydelliseksi testaamiseksi on tärkeää että sama henkilö joka on kirjoittanut koodin myös testaa sen. Täten voidaan varmistua siitä että kaikki mahdolliset koodipolut ovat tulleet testatuksi. (Kassem s.224–226, STF)

4.5 TDD

TDD eli Test Driven Development tunnetaan myös Test First Development:na eli “testi ensin kehitys”. TDD:ssä kirjoitetaan testit ensin eli kirjoitetaan testit vaatimuksista jotka on koodiin suunniteltu. Tiedettäessä liiketoiminnan vaatimukset ohjelmalle, ne

toteutetaan testeiksi. Ennen kuin testit saadaan onnistuneesti suoritettua, ei voida sanoa että ohjelma vastaa vielä sille asetettuja vaatimuksia.

Aluksi testin kirjoittamisen jälkeen ohjelman ei pitäisi edes kääntyä. Tämän jälkeen on kirjoitettava luokat ja mahdollisimman yksinkertainen koodi, jolla saadaan testi suoriutumaan onnistuneesti. Tämän jälkeen kirjoitetaan uusia testejä vaatimusten mukaisesti kunnes kaikki vaikeimmatkin testit on onnistuneesti suoritettu.

Hyötynä TDD:ssä on varmistaa koodin laadun alusta alkaen ja kehittäjät kirjoittavat ainoastaan tarvittavat koodit jotta vaatimuksiin päästäisiin. Tällä tavoin voidaan myös varmistaa että syötteet ja lopputulokset ovat varmasti oikeita ja haluttuja. (Bender & McWherter s.8-10)

5 MUUT TESTAUSVAIHEET

5.1 Integraatiotestaus

Integraatiotestaus on looginen jatkumo yksikkötestaukselle. Tyypillisimmillään integraatiotestissä on yksi komponentti jota toinen komponentti kutsuu. Kokonaisuuden toimintaa pitäisi myös testata, jotta voidaan varmistua miten eri yhdistelmät toimivat keskenään.

Integraatiotestauksen tarkoituksena on paljastaa virheitä, jotka syntyvät kun yksittäisiä komponentteja yhdistetään ja testataan ryhmässä. Usein testauksessa käytetään apuna stubitusta. On syytä myös varmistaa että kaikki on yksikkötestattu ennen varsinaisen integraatiotestausten tekemistä. (Graham & Veenendaal s.42–43, Haas s.12)

5.2 Järjestelmätestaus

Järjestelmätestauksessa koko toteutettua järjestelmää tai järjestelmän osaa testataan. Testauksessa käytetään yleensä hyväksi Black Box – testausta. Testauksen suorittavat yleensä testaaja – nimikkeellä olevat henkilöt. (STF)

Järjestelmätestin tarkoituksena on testata vastaako toteutettu järjestelmä sille asetettuja vaatimuksia. Testaajan löytäessä virheen järjestelmästä tehdään havainto jonka

perusteella toteuttaja tekee korjauksen ja jonka testaaja testaa uudelleen korjauksen saapuessa järjestelmätestiin.

Järjestelmätestin tulisi olla mahdollisimman kattava. Sen pitäisi kattaa käyttäjätapaukset, liiketoimintavaatimukset, tekniset vaatimukset sekä suorituskyky vaatimukset. (Haas s.14–15)

5.3 Regressiotestaus

Regressiotestauksella varmistetaan että, oli se sitten ohjelman parantaminen tai virheen korjaus, ohjelmistomuutos ei riko järjestelmää vaan se toimii edelleen vaatimusten ja määrittelyjen mukaisesti. Regressiotestauksen aikana ei luoda uusia testitapauksia vaan käytetään olemassa olevia testejä uudelleen.

Regressiotestausta voidaan suorittaa kaikilla testauksen eri tasoilla mutta relevantein käyttötarkoitus sille on järjestelmätestaus. Testit ovat usein aikaa/resursseja vieviä, jonka vuoksi yhä enemmän regressiotestausta suoritetaan automaatiotestausvälineillä.

Regressiotestit olisi pyrittävä pitämään niin vähäisinä kuin mahdollista kuitenkaan vähentämättä testattavan alueen kattavuutta. Hyvänä tapana voidaan pitää että jokaisesta korjatusta virheestä kirjoitetaan testi, jos löytyy useampi testi samalle asialle, pitäisi vähemmän tehokkaasta testistä hankkiutua eroon. (Graham & Veenendaal s.49–50, MSDN)

5.4 Savutestaus

Savutestaus on käännöksen varmistustestaus. Testin tarkoituksena on varmistaa että ohjelman tärkeimmät ominaisuudet toimivat. Savutestauksen tulosten perusteella voi päätellä onko seuraavalle testaustasolle siirtyminen järkevää. Yleensä jos käännöksiä on usein, on viisasta automatisoida savutestaus käännöksen yhteyteen. Savutestausta käytetään tyypillisesti integraatio-, järjestelmä- sekä hyväksymistestauksen tasoilla. (Craig & Jaskiel s.129–130)

5.5 Hyväksymistestaus

Hyväksymistestaus on testauksen taso, jossa järjestelmä testataan hyväksyttäväksi. Tämän testauksen tarkoitus on täsmentää vastaako järjestelmä liiketoiminnan vaatimuksia sekä vastaako toimitettu tuote haluttua lopputulosta. (Haas s.15)

Yleensä hyväksymistestauksessa käytetään Black Box – testaus menetelmää. Jokainen hyväksymistesti esittää jotain haluttua lopputulosta järjestelmältä. Hyväksymistestausta käytetään myös regressiotestauksena ennen varsinaista julkaisua. (STF)

Hyväksymistestaus suoritetaan järjestelmätestauksen jälkeen ja ennen ohjelman siirtoa varsinaiseen käyttöön. Yleensä testauksen suorittaa projektiin kuulumattomat samassa organisaatiossa olevat henkilöt tai asiakkaan oman organisaation henkilöt. (Haas s.16)

6 AUTOMAATIOTESTAUS

6.1 Miksi testejä automatisoidaan

Testien automatisointi helpottaa järjestelmätestausta. Automaatiotestit ovat pääsääntöisesti regressiotestausta, jossa testataan että aikaisemmin toimineet ominaisuudet toimivat yhä ohjelmamuutosten jälkeenkin.

Testit ovat helpommin toistettavissa ilman käyttäjän mahdollisesti tekemiä virheitä. Testeillä on helppo simuloida suuria määriä syötteitä ja niiden lopputuloksia. Testauksesta vaaditaan jatkuvasti yhä nopeampaa ja testien automaatiosoinnilla saadaan vähennettyä merkittävästi hidasta manuaalista työtä.

Automaatiotestaus ei vähennä ohjelman virheitä jos manuaaliseen työntekoon ei riitä rahat eivätkä resurssit. Automaatiotestauksen kohteet on valittava huolella ja tarkkaan mitä testejä todella kannattaa automatisoida. Testien kirjoittamisessa on oltava myös tarkkana sillä väärin tehty automaatiotesti voi tulla lopulta todella kalliiksi.

Automaatiotestaus tuo kolme selvää hyötyä joita ovat virheiden jatkuva seuranta, löytyminen ja kustannusten pienentyminen virheiden löytyessä aikaisessa vaiheessa, ma-

nuaalisten toistojen määrä vähenee joka säästää aikaa ja vähentää kustannuksia sekä pystytään parantamaan tuottavuutta. (Hayes s.8-14)

6.2 Milloin testejä ei pidä automatisoida

Jos ohjelmassa on paljon reaaliaikaista tai muuttuvaa dataa, jota ei voida etukäteen ennustaa, on parempi suorittaa testaus manuaalisesti. Lisäksi jos testiympäristö ja testidata eivät ole hallinnassasi on mahdotonta tehdä automaatiotestejä. Tilanne tietokannassa voi muuttua hallitsemattomasti jolloin testit auttamatta muuttuvat käyttökelvottomiksi. (Hayes s.15–17)

7 KÄYTTÖLIITTYMÄN AUTOMAATIOTESTAUS

7.1 Testauksen vaatimukset

Käyttöliittymä vastaa käyttäjän tekemiin tapahtumiin kuten hiiren liikkeeseen sekä valikoiden käyttämiseen. Käyttöliittymä reagoi ja suorittaa toimintoja sen takana olevan koodin avulla, metodien sekä viestien kautta. Käyttöliittymät tekemät järjestelmistä helppokäyttöisiä ja vaativat yhä enemmän kehittäjän aikaa ja panosta. Käyttöliittymää perinteisesti testataan nauhoituksilla jotka generoivat käyttäjän hiiren liikkeet sekä näppäimistön painallukset. Testaustyökalut nauhoittavat kaikki tapahtumat yleensä skripteihin tai niitä vastaaviin metodeihin. Myöhemmin testiä ajettaessa se toistetaan käyttäjälle täsmälleen, kuten nauhoitus on tehty, jos ohjelma ei kykene suoriutumaan annetuilla syöte- ja tarkastusarvoilla testi epäonnistuu. (Memon s.87-88)

7.2 Testausmenetelmät

Käyttöliittymän automaatiotestaus voidaan jakaa kolmeen osaan. Ensimmäisessä osassa päätetään mitä aiotaan testata, toisessa vaiheessa toteutetaan haluttu testi ja viimeisessä vaiheessa testataan käyttöliittymän tiloja ja tuloksia suoritetun testin jälkeen. Käyttöliittymällä testataan tyypillisesti objektien tiloja ja ominaisuuksia. Testeissä testataan mm. nappien, syötetietojen sekä tekstikenttien tiloja sekä ominaisuuksia kuten taustavärejä, fontteja sekä syötetekstejä ja -arvoja. Kaikilla käyttöliittymän objekteilla

on tiettyjä ominaisuuksia joita voidaan tutkia ja näin saadaan tieto miten käyttöliittymä toimii kussakin tilanteessa. (Last, Kandel & Bunke s.52-65)

Automatisoitujen käyttöliittymätestien tarkoituksena on helpottaa testaajan työtä, vähentää manuaalisten testien määrää sekä jatkuvasti testata käyttöliittymän ominaisuuksia jos jokin on muuttunut ei halutulla tavalla ohjelman kehityksen aikana. (Memon s.87-88)

8 MICROSOFT CODED UI – VÄLINE

8.1 Yleistä

Microsoftin Coded UI – työkalu tuli Visual Studio 2010 versioon Feature pack 2:n mukana. Työkalun tarkoituksena on mahdollistaa käyttöliittymän automaatiotestien tekeminen suoraan Visual Studiolla. Työkalulla voi luoda useita erityyppisiä testejä käyttäen joko Visual Studion Premium tai Ultimate versioita, muilla Visual Studion versioilla ei ole mahdollista tehdä automaatiotestejä. Automatisoidut testit ajavat testi steppejä ja määrittelevät onnistuuko vai epäonnistuuko testi. Näin testejä voi ajaa nopeammin ja useammin kuin manuaalisessa testauksessa. Automatisoidut testit testaavat nopeasti toimiiko ohjelma yhä siihen tehtyjen koodimuutosten jälkeen.

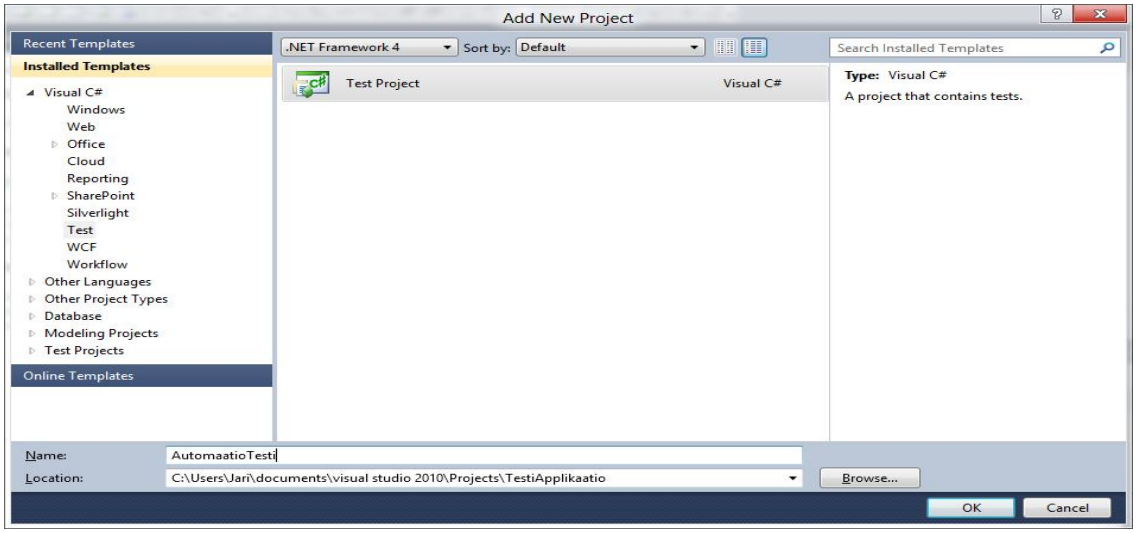
8.2 Testaustyökalun asentaminen

Asentamiseen tulee ladata Microsoftin sivuilta Feature Pack 2. Asentaminen on yksinkertainen toimenpide. Asennustiedoston latauduttua se käynnistetään kuviossa 5 olevasta Next - painikkeesta ja paketti tulee automaattisesti Visual Studioon valmiiksi käytettäväksi.

KUVIO 5 Feature Pack 2 asentaminen

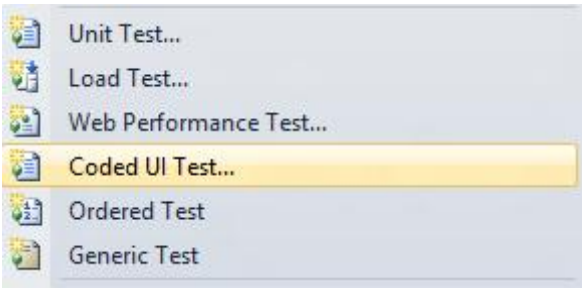
8.3 Projektin ja testin luominen

Kuviossa 6 näytetään miten projekti tehdään luomalla uusi projekti ratkaisu (solution).



KUVIO 6 Projektin luominen

Tämän jälkeen lisätään projektiin Coded UI Test kuten kuviossa 7 näytetään, jonka jälkeen Visual Studio lisää tarvittavat referenssit ja olet valmis tekemään testin.



KUVIO 7 Projektin valinta

8.4 Testien rakenne

Kuviossa 8 näytetään testin perusrakenne:

```
public class CodedUITest1
{
    public CodedUITest1()
    {
    }

    [TestMethod]
    public void CodedUITestMethod1()
    {
        // To generate code for this test, select "Gener
        // For more information on generated code, see h
    }

    Additional test attributes

    /// <summary>
    ///Gets or sets the test context which provides
    ///information about and functionality for the curre
    ///</summary>
    public TestContext TestContext
    {
        get
        {
            return testContextInstance;
        }
        set
        {
            testContextInstance = value;
        }
    }
    private TestContext testContextInstance;
}
```

KUVIO 8 Testin perusrakenne

Coded UI – testin luonnin jälkeen Visual Studio tekee automaattisesti tarvittavat tiedostot automaatiotestiä varten.

Testin nauhoittamisen jälkeen Studio luo UIMap.uitest – tiedoston ja sen alitiedostoiksi UIMap – koodi sekä – designer tiedostot.

UIMap – luokka sisältää jokaisen metodin koodin, jotka on määritelty nauhoitusta luodessa. UIMap:n designer – tiedostoon ei saa tehdä muutoksia koska muutokset voivat kadota. UIMap.uitest on XML kartta testistä. UI Actions on ikkuna, jossa nauhoitettujen metodein askelmat ja assertointi – metodit. Properties -ikkunalta pääset tarkemmin katsomaan askelmien ominaisuuksia ja määrittelemään mm. sen jos kyseistä askelmaa (step) ei löydykään, kaatuuko ohjelma virheeseen vai jatkaako ilman virhetilannetta. Tällaisia tapauksia voivat olla muun muassa messagebox – viestit jotka eivät välttämättä jokaisen ajon aikana välttämättä tule ilmentymään. Haasteena on löytää tapaus, jossa kaikki nämä eri tilanteet saisi tulemaan. Jos seuraavalla ajokerralla tuleekin jokin ponnahdusikkuna jota ei edellisessä nauhoituksessa ole tullutkaan, testistä tulee virheilmoitus, vaikkei se ole haluttu lopputulos.

Oikealla puolella on UI Control Map – ikkuna jossa näkyy kontrollien sijainnit ja nimet. Properties – ikkunalta pääsee tarkastelemaan lähemmin kontrollin ominaisuuksia.

8.5 Testien suunnittelu

Testit kannattaa suunnitella siten että ne ovat mahdollisimman helppo ylläpitää. Pitkiä testejä ei kannata ruveta nauhoittamaan sillä jos jokin testissä muuttuu, voit joutua nauhoittamaan testin uudelleen. Etukäteen mahdollisuuksien mukaan on tärkeää miettiä, minkälaisiin palasiin testi tulisi jakaa. Yksi kokonaisuus kannattaa sijoittaa yhteen testiin, koko ohjelmaa ei kannata missään tapauksessa testata yhdellä tai kahdella testillä. Jos jokin testattava kohta muuttuu, on se helppo nauhoittaa uudelleen paloiteltuun testiin. Jokaisen uuden sivun tai näytön tulisi olla oma testinsä. Testin nimeäminen on ehdottoman tärkeää jotta tietää käyttöliittymän muuttuessa, mikä testi tulisi korvata uudella ja järjestys pysyisi oikeana.

Testien tarkistukset (assertointi) kannattaa myös miettiä missä vaiheessa niitä tekee. Tarkistusten kohdalla on katkaistava nauhoitus ja tehtävä siitä metodi ja tämän jälkeen täytyy jatkaa nauhoitusta uudella metodilla.

8.6 Testien datalähteet

Testiin voidaan lisätä datalähde, jolloin saadaan tehtyä useampi suorituskerta yhden testin pohjalta. Testin luonnin jälkeen tietyllä datalla, voi testiä ajaa useita kertoja eri arvoilla lisäämällä jonkin datan josta arvoja luetaan. Data -lähteeseen voi lisätä erilaisia parametreja joiden avulla ohjelmaa testataan. Jokainen rivi datasta on oma suorituskertansa testistä. Kokonaistulos on näiden suorituskertojen summa, eli jos yksikin suorituskerta epäonnistuu, epäonnistuu tällöin koko testi. Eri datalähteinä voidaan käyttää tietokantoja, XML – tiedostoja tai CSV – tiedostoja. Testiesimerkki löytyy liitteestä 1.

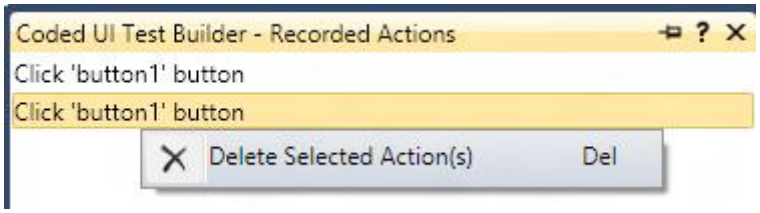
8.7 Testien nauhoittaminen

Uusi testi nauhoitetaan valitsemalla hiiren oikealla painikkeella testimetodin päällä ja painamalla Use Coded UI Test Builder nappulaa kuten kuviossa 9 näytetään.

KUVIO 9 Testin nauhoittaminen

Tämän jälkeen ilmestyy laatikko näytön alareunaan josta valitaan punainen ympyrä joka aloittaa nauhoituksen. Filmin kohdalta voidaan katsella ja muokata nauhoitettuja askelmia. Tähtäimen kohdalta voidaan tarkistaa (assertoida) haluttua kohtaa. Paperin kuvasta päätetään nauhoituksessa ollut testi. Huomioitavaa testin nauhoituksessa on että debug – modessa ei voi nauhoituksia tehdä.

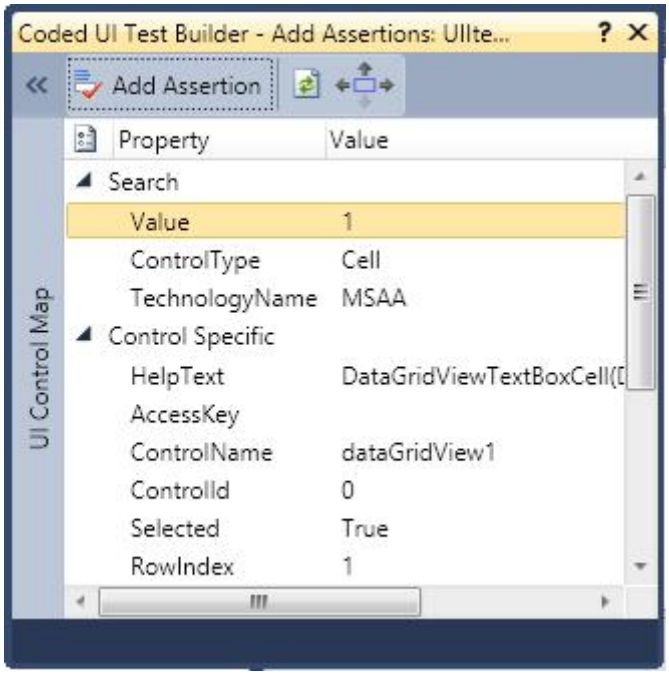
Filminauha painiketta painamalla voit tarkastella ja muokata luomaasi nauhoitusta. Voit poistaa haluamasi painallukset kuvion 10 mukaisesti jos et halua säilyttää niitä lopullisessa testissä.



KUVIO 10 Nauhoituksen muokkaaminen

8.8 Testien validointi

Testien tarkistaminen (validointi) tapahtuu tähtäimen kuvasta. Kontrolli tarkistetaan viemällä kohdistin siihen kohtaan joka halutaan tarkistaa. Kohdistin vapautetaan halutussa kohdassa esim. taulukon solussa jolloin voidaan valita, mikä arvo solusta tarkistetaan. Kontrolleista voidaan tarkistaa kuvion 11 mukaisesti mm. arvoja, kontrollintyyppiä, kontrollin nimeä, onko se valittuna, rivi-indeksiä yms.

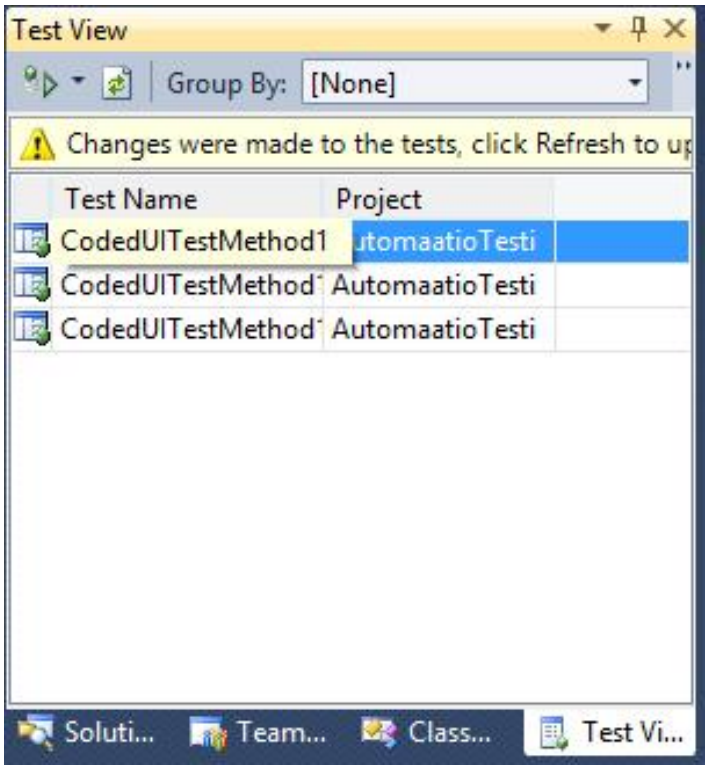


KUVIO 11 Kontrollin tarkistus

Tämän jälkeen painetaan ”Add Assertion” – nappulaa, jonka jälkeen testin pääluokkaan lisätään tarkistus – metodi.

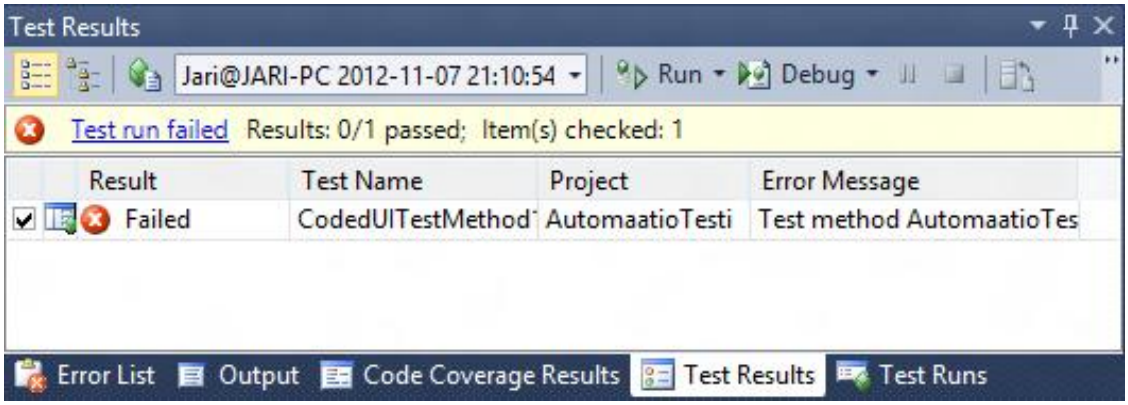
8.9 Testien ajaminen

Testejä voidaan ajaa kuten tavallisia yksikkötestejä, koodista käsin tai kuviossa 12 olevalta Test View – ikkunalta. Testejä voidaan ajaa joko normaalissa tai debug –tilassa. Debug –tilassa päästään seuraamaan testien tapahtumia koodirivi kerrallaan. Testien tapahtumat ja lopputulokset näkyvät samaan tapaan kuin yksikkötesteissäkin.



KUVIO 12 Test View –näkymä

Testien lopputuloksia voi tarkistella yksikkötestien tapaan kuviossa 13 olevalta Test Results – näkymältä.



KUVIO 13 Test Results -näkymä

8.10 Testien ylläpito

Testien ylläpito Coded UI – välineellä on haasteellista. Kaikki lähtee testien nauhoittamisesta, jossa pitää huomioida että testejä saatetaan joutua ylläpitämään jossain vaiheessa. Tässä vaiheessa onkin viisainta tehdä nauhoituksista mahdollisimman lyhyitä ja ytimekkäitä. Testejä kannattaa tehdä esim. yksi alinäkymä kerrallaan, taikka testejä voidaan myös myöhemmin nauhoituksen jälkeen jakaa useampiin metodeihin ”Split to a new method”.

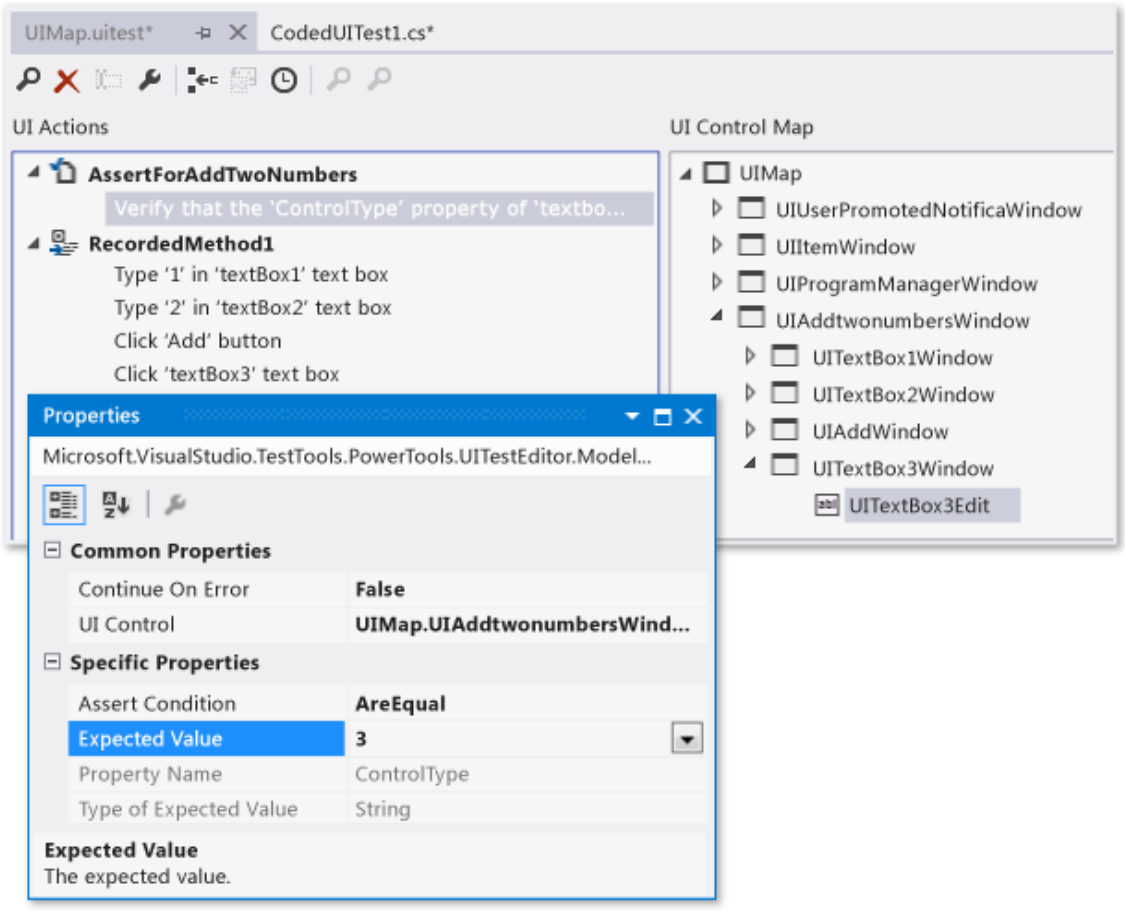
Ylläpitovaiheen kohdatessa kannattaa katsoa läpi aikaisempi nauhoitusta ja tutkia kooditasolla mitä testillä on haluttu testata ja onko siinä tarkistuksia. Tarkistukset ovat omia metodejaan, joten niihin ei välttämättä tarvitse mennä koskemaan lainkaan. Kannattaa poistaa turhaksi jäävä metodi vasta kun uusi on nauhoitettu, jotta saadaan vertailtua vanhan ja uuden metodin välillä että kaikki tulee varmasti testattua.

Uusi nauhoitus tehdään normaaliin tapaan ja nauhoituksesta tehdään metodi. Tähän metodiin sitten viitataan halutusta testistä mihin muutos kohdistuu. Kannattaa myös pitää mielessä että olemassa olevista nauhoituksista pystyy muuttamaan esim. tarkasteltavia arvoja, syötearvoja tai haluttua lopputulosta. Tämä voi olla jossain tilanteissa mielekkäämpää kuin koko testin uudelleen nauhoitus. Tietoja muuttaessa suoraan nauhoitettuun metodiin, täytyy olla tarkkana että muuttaa tiedot kaikkiin tarvittaviin

kohtiin. Arvoja saattaa löytää designer – tiedostosta sekä luokasta jossa metodin toteutus on.

8.11 Visual Studio 2010 vs. Visual Studio 2012

Visual Studio 2012 Coded UI – työkaluun on lisätty paljon uusia ominaisuuksia verrattuna vanhempaan versioonsa. Nyt tuettujen selaimien joukossa on myös IE 9 & 10. Näille selaimille löytyy myös tuki HTML5:lle. Testien konvertointi vanhasta versiosta uuteen versioon tapahtuu automaattisesti ja editori on muuttunut täysin erinäköiseksi kuten kuviosta 14 voi nähdä. Muita parannuksia on esim. kuvausten kirjoittaminen nauhoitettaviin testeihin.



KUVIO 14 Visual Studio 2012 (MSDN)

8.12 Muita automaatiotestausvälineitä

Automaatiotestaustyövälineitä on olemassa runsaasti. Avoimen lähdekoodin työkaluista voitaisiin maintia Selenium joka on suunnattu verkkosivujen automaatiotestaukseen. Kaupalliselta puolelta Test Automation FX on Visual Studio 2010 ja Visual Studion 2008 –versioille suunnattu testaustyökalu.

9 TULOKSET

Tutkimuksen tarkoituksena oli tutkia onko työkalusta hyötyä testauksen automatisoinnissa. Tutkimustulokset viittaavat siihen että kysyntää työkalulle on ja sen käyttöönotto keventäisi testausprosessia ja helpottaisi testaajien työtä. Rutiininomaisesta manuaalitestaamisesta päästäisiin eroon sitä mukaan kun testejä luotaisiin. Käyttöliittymän automaatiotestit takaavat tasaisen laadun ja inhimillisten virheiden vaikutus vähenee huomattavasti koska testi toistuu joka kerta samanlaisena. Tämä työkalu ei kuitenkaan tule korvaamaan manuaalista testaamista vaan antaisi tukea ja helpottaisi regressiotestausta sekä säästäisi paljon työaikaa.

Epäselväksi vielä tutkimuksen jälkeen jäi kuin hyvin työkalu toimii päivittäisessä käytössä. Tämä selviää vasta kun työkalua on testattu ohjelmassa pidemmän aikaa ja saadaan vertailupohjaa kuinka paljon tämä todellisuudessa säästäisi työaikaa. Laajamittainen käyttöönotto vaatisi paljon aikaa ja resursseja käyttöönsä.

Suurimmaksi ongelmaksi työkalun käytössä muodostuu testien ylläpidettävyys. Ohjelmat ja niiden vaatimukset muuttuvat jatkuvasti kun uusia ominaisuuksia tulee ja olemassaolevia korjataan ja näin ollen testeistä voi piankin tulla epävalideja. Ajan kuluessa vasta selviää muodostuuko testien ylläpidettävyydestä onglemia. Ylläpidettävyyden ongelmat ovat ratkaistavissa oikeanlaisella suunnittelulla ja käyttäjien riittävällä koulutuksella. Kokonaisuutta tarkasteltaessa onkin syytä ottaa huomioon, onko työkalu riittävän monipuolinen ja palveleeko se tarkoitustaan työmäärien pienentämisessä.

Testien tekeminen vaatii hieman testaus- sekä ohjelmointiosaamista. Ilman ohjelmointiosaamista sekä ymmärtämistä ei työkalulla käytännössä voi tehdä kovin laajamittais-

ta ja kattavaa automaatiotestausta. Tämä on korvattavissa testaajan ja toteuttajan tiiviillä yhteistyöllä.

Testeissä kannattaisi keskittyä pieniin palasiin kerrallaan jotta ne olisivat helposti ylläpidettäviä. Massiiviset testit täytyy melko suurella todennäköisyydellä nauhoittaa ohjelmaan tulevan muutoksen jälkeen kokonaan uudelleen. Testin jakaminen pienempiin osiin helpottaakin testien ylläpitoa sillä ohjelman muuttuessa ei tarvitse nauhoittaa uudelleen kuin pieni palanen testiä.

10 POHDINTA

Opinnäytetyön teoria ja käytäntö tukevat hyvin toisiaan sillä ilman testauksen laajamittaista ymmärtämistä on vaikea lähteä tekemään testejä. Testauksen kokonaisuuden hahmottaminen on mielestäni edellytys onnistuneille automaatiotesteille.

Testaajien ja kehittäjien välinen kommunikointi on äärimmäisen tärkeää. Testit kannattaa tehdä yhteistyössä testaajien ja kehittäjien välillä sillä tällöin testaajat voivat kertoa mitkä ohjelmiston osa-alueet kannattaa suorittaa automaatiotestauksena.

Käyttöliittymän automaatiotestaus on aiheena laaja ja testaustyövälineitä löytyykin runsaasti. Valitsin yhden työkalun tarkempaan tarkasteluun, joka on Microsoft Visual Studion Coded UI –työkalu koska se on jo valmiiksi valmiiksi saatavilla Microsoftin Visual Studioon eikä tarvitse erikseen miettiä kolmannen osapuolen lisenssiehtoja ja työkalulta ei tule loppumaan tuki, joka voi olla riskinä monen pienemmän yrityksen ohjelmiston kohdalla. Tutkimus eteni suunnitelmien mukaan lukuun ottamatta aikataulullisia viivästyksiä. Toimeksiantajan puolelta ei ollut aikataulua työn valmiiksi asti saattamisessa.

Teoria osuus oli hyvin avartava ohjelmistokehittäjän näkökulmasta. Testauksessa on paljon eri vaiheita ja asioita jotka tulisi ottaa huomioon, joita ei tavallisessa perustyössään kehittäjänä tule edes ajatelleeksi. Testauksen eri vaiheet ja niiden tarkoitukset tulisikin olla jokaisen ohjelmistokehittäjän tiedossa.

Ongelmaksi testeissä voi muodostua se että automaatiotestit menevät aina tietokantaan asti. Testien jälkeen ei pysty tekemään tietokannan palautusta ohjelmallisesti.

Tietokannan palauttaminen alkuperäiseen tilaan varmuuskopiosta testien suorittamisen jälkeen poistaisi tämän ongelman.

Visual Studion apuväline on tällaisenaan vielä kankea ja taipumaton muutosten tekemiseen. Jos yrityksellä on suunnitelmissa ottaa automaatiotestausta käyttöön, kannattaa tällöin katsoa myös muita automaatiotestauksen työvälineitä joista voisi saada suuremman hyödyn irti työssään.

Automaatiotestauksen tekeminen vie paljon aikaa ja toisen henkilön tekemien testien ylläpitäminen on hankalaa. Hyötynä käyttöliittymän automaatiotestauksessa on testien tehokkuus. Kustannukset voidaan kuitenkin saada takaisin moninkertaisesti ajan kuluessa. Ylläpidosta saattaa tulla joitain lisäkustannuksia mutta oikein tehtynä ja käytettynä automaatiotesteillä voidaan saavuttaa suuri taloudellinen hyöty ja testaajan ajan voi säästää muihin testitapauksiin kuin käyttöliittymän regressiotestauksen tekemiseen.

LÄHTEET

Bender J, McWherter J. 2011. Professional Test Driven Development with C#: Developing Real World Applications with TDD Viitattu 1.10.2012. Http://books.google.fi.

Burnstein I. 2003. Practical Software Testing: A Process-Oriented Approach Viitattu 1.10.2012. Http://books.google.fi.

Codeproject Advance Unit Testing 2003. Viitattu 4.12.2012 http://www.codeproject.com/Articles/5019/Advanced-Unit-Testing-Part-I-Overview

Craig R D, Jaskiel S P. 2002. Systematic Software Testing Viitattu 1.10.2012. Http://books.google.fi.

Defect Management 2009. Viitattu 4.12.2012 http://www.testingisuseful.com/defectmanagement.htm

Graham D, Veenendaal E. 2008. Foundations of Software Testing: ISTQB Certification Viitattu 1.10.2012. Http://books.google.fi.

Hass A. 2002. Guide to Advanced Software Testing Viitattu 1.10.2012. Http://books.google.fi.

Hayes L. 2011. The Automated Testing Handbook Viitattu 1.10.2012. Http://books.google.fi.

Kassem A.S, Gopalawswamy R. 2009. Software Engineering Viitattu 1.10.2012. Http://books.google.fi.

Last M, Kandel A, Bunke H. 2004. Artificial Intelligence Methods in Software Testing Viitattu 28.11.2012 Http://books.google.fi.

Limaye. 2011. Software Testing Viitattu 1.10.2012. Http://books.google.fi.

Luo L. Software Testing Techniques Viitattu 2.10.2012 http://www.cs.cmu.edu/~luluo/Courses/17939Report.pdf

Memon A. 2002,GUI Testing: Pitfalls and Process. Viitattu28.11.2012 http://www.cs.umd.edu/~atif/papers/MemonIEEEComputer2002.pdf

MSDN Library . 2012, Viitattu 1.10.2012 http://msdn.microsoft.com/en-us/library

Pan J. 1999. Software Testing Viitattu 1.10.2012. http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/

Software Process Models. Viitattu 1.10.2012 http://www.the-software-experts.de/e_dta-sw-process.htm

Software Testing Fundamentals. 2012, Viitattu 1.10.2012 http://softwaretestingfundamentals.com/

Testaussuunnitelma. 2001, Viitattu 4.12.2012 http://www.soberit.hut.fi/tik-76.115/00-01/palautukset/groups/Osprey/t4/testaussuunnitelma.html

LIITE 1

Datalähteet:

CSV

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
"|DataDirectory|\\data.csv", "data#csv", DataAccessMethod.Sequential), Deployment-
Item("data.csv"), TestMethod]
```

Excel

```
DataSource("System.Data.Odbc", "Dsn=Excel Files;Driver={Microsoft Excel Driver
(*.xls)};dbq=|DataDirectory|\\Data.xls;defaultdir=.;driverid=790;maxbuffersize=2048;
pagetimeout=5;readonly=true", "Sheet1$", DataAccessMethod.Sequential), TestMeth-
od]
```

Test case in Team Foundation Server

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.TestCase",
"http://vlm13261329:8080/tfs/DefaultCollection;Agile", "30", DataAccessMeth-
od.Sequential), TestMethod]
```

XML

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML",
"|DataDirectory|\\data.xml", "Iterations", DataAccessMethod.Sequential), Deploy-
mentItem("data.xml"), TestMethod]
```

SQL Express

```
[DataSource("System.Data.SqlClient", "Data Source=.\\sqlexpress;Initial Cata-
log=tempdb;Integrated Security=True", "Data", DataAccessMethod.Sequential), Test-
Method]
```

Esimerkki käytöstä

```
[DeploymentItem("DataDriven.csv"),
   DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
      "|DataDirectory|\\DataDriven.csv", "DataDriven#csv",
      DataAccessMethod.Sequential),
```

```
    TestMethod]
public void CodedUITestMethod1()
{
    // To generate code for this test, select "Generate Code for
    // Coded UI Test" from the shortcut menu and select one of
    // the menu items.
    this.UIMap.AddTwoNumbersParams.TextInput1EditText =
        TestContext.DataRow["Input1"].ToString();
    this.UIMap.AddTwoNumbersParams.TextInput2EditText =
        TestContext.DataRow["Input2"].ToString();
    this.UIMap.AddTwoNumbers();

    this.UIMap.AssertforAddExpectedValues.TextAnswerEditText =
        TestContext.DataRow["ExpectedResult"].ToString();
    this.UIMap.AssertforAdd();
}
```